文章结构

1. 题目：.....1
2. 分析：.....1
3. 代码与注释：.....1
4. 附录：.....5
4.1. 算法原理：.....5
4.1.1. 串行基数排序步骤如下：.....5
4.1.2. GPU 基数排序原理：.....5
4.1.3. Block 内有序序列归并：.....6

1. 题目：

非常多的（上亿个）数字组成的数组，请利用 GPU 实现排序。

2. 分析：

数字的数量庞大带来的第一个难点是：需要考虑显存大小，即需要考虑分片排序后再归并。

数字数量的庞大第二个难点是：如果采用递归类算法，如：快速排序和归并排序，无疑以 GPU 的寄存器大小，一定会产生栈溢出。所以必须采用非递归排序算法。其中根据 CUB 库的作者所进行的比较实验显示，基数排序无疑是最适合实现 GPU 多线程排序的排序算法。基于以上分析，将利用基数排序实现 GPU 的算法加速。

3. 代码与注释：

```
    /*利用 GPU 对非常多的数据进行排序*/


    //所有进行基数排序的数据都需要转化为无符号的 32 位整型。以方便之后的位计算。
    typedef unsigned int ubit32;

    #define BLOCKSIZE = 512


    /*
    GPU 的基数排序分成两个步骤，
    1、线程内的基数排序：每个线程负责自己部分的数据，然后进行基数排序，线程负责部分内部数据成为从小到大的有序序列。
    2、线程间有序序列归并：将每个线程负责的已经有序的序列按照有序序列归并的方式，block 中所有线程每次只都找到一个最小值，然后从小到大插入到结果序列中。
    */
    __device__ void radix_sort(
                            ubit32 *data,
                            ubit32 *tmp,
                            unsigned int sizeperthrd,
                            unsigned int blockdimx,
                            unsigned int len,
                            unsigned int tid
                        )
    {
    //基数排序方法：
    //1、每次只取出序列中候选待比较数的当前位。
    //2、将此位为 0 的值放入表示序列前半部分的序列(bucket0)中。
    //3、将此位为 1 的值放入后半部分的序列中。
    //4、序列前半部分与后半部分合并，形成新的序列。
    //5、当前位比较完成，比较更高的位。
        for(unsigned int bit = 0; bit < 32; bit++)
        {
         //bit_mask 为取位掩模，它只会有在某一位上为 1，那么任何二进数与之相与操作。只会将原数某一位上是
```

```
0 或 1 给提取出来。
        //掩模提取候选数字相应位的作用是根据其是 0 还是 1 来判断此数当前循环下放入序列的前半部分还是后半部分。
        ubit32 bit_mask = 1 << bit;
        unsigned int cnt0 = 0;
        unsigned int cnt1 = 0;
        //申请共享内存空间，反复读写共享内存的速度比反复读取全局内存速度要快很多。
        //这里考虑到片内共享内存空间的限制，每个 block 处理的数据不能太多，即 len 不能太大。
        //假设一个 block 包含 512 个线程，每个线程处理 sizeperthrd 个数，那么 len 必须小于等于 512*sizeperthrd。
        __shared__ unsigned int bucket0[len];  //利用共享内存作为 block 内各线程都能访问的前半个序列。
        __shared__ unsigned int bucket1[len];  //同样利用共享内存实现后半部分。
        //每个线程负责一部分的数据，这里线程不负责相邻数据，原因是会造成线程非合并访问：
        //      即当线程 1 需要取得指针 0 位置数据时，线程 2 需要取指针 32 位置的数据，那线程 2 只能等线程 1 取完后，才能取指针 32 处的数。造成速度过慢。
        //      所以,线程 1 取 0 位置、32 位置、64 位置....线程 2 取 1 位置、33 位置、65 位置...线程 3.....
        //      这样的取数方式会使得全局内存访问的效率大大提升。
        for(unsigned int i = 0; (i < len) && (i + tid < len); i += sizeperthrd)
        {
            // 该循环每个线程只负责将自己负责的线程中的数进行基数排序。
            // sizeperthrd 如果为 32，则表示，tid=0 的线程每次循环只会去拿一个数，每次取得的数在地址上相差 32，
            // i 表示第 i 次循环。

            //将本线程待排序的候选值取出。
            ubit32 value = data[i+tid];
            //候选值与掩模相与，只有保留 value 某一位的值（是 0 或者 1），而这个值如果是 0，将会被放入新序列的前半部分。
            //否则将被放入到后半部分。
            if((value&bit_mask)>0)
            {
                //每个线程都会把自己的候选值放入至相隔 sizeperthrd 的空间中。tid 线程号与累加 sizeperthrd 的计数器 cnt1 负责指示结果存放位置。
                bucket1[cnt1+tid] = value;
                cnt1 += sizeperthrd;
            }
            else
            {
                bucket0[cnt0+tid] = value;
                cnt0 += sizeperthrd;
            }
        }
        for(unsigned int i=0; i<cnt1; i+=sizeperthrd)
        {
            bucket0[cnt0 + i + tid] = bucket1[i+tid];
        }
        }
        __syncthreads();
        for(unsigned int i=0;  (i < len) && (i + tid < len); i+=sizeperthrd)
        {
        //将结果放入中间缓存的 tmp 中，供之后的并行归并操作。
        tmp[i+tid] = bucket0[i+tid];
        }
    }

    __device__ void mergeparallel(
                                //*data 为并行归并的输出空间，并行归并操作最终将结果输出至*data 所指空间中
                                    ubit32 *data,
                                    //*tmp 为并行归并的输入空间，也是整个基数排序的中间缓存空间。
                                    ubit32 *tmp,
                                    //len 为为当前 block 下处理的数据的大小。
```

```
                                unsigned int len,
                                //tid 为当前的线程编号。
                                unsigned int tid,
                                //threadnum 为一个线程需要负责排序的数据个数。
                                unsigned int threadnum
                            )
    {
    // minValue 用来存放当前 block 中所有线程中最小值。
    __shared__ ubit32 minValue;
    // mintid 用来存放当前 block 中取得最小值的那个线程编号。
    __shared__ unsigned int mintid;
    // tidHeader 用来存放 thread 负责候选数据的头指针偏移量，
    // 该偏移量也表示当前线程负责的数据部分中，之前已经有多少个值被上报为最小值，并被取走了。
    // 那么下次该 thread 就从该头指针开始取候选值，与其他线程进行最小值比较。
    __shared__ unsigned int tidHeader[threadnum];
    unsigned int value = 0;
    tidHeader[tid] = 0;
        __syncthreads();
        //循环 i 次，每次都会找到 block 内所有线程负责的数据中，最小的那个，而 block 内有 len 个数，那么
一共要循环 len 次。
        for(unsigned int i=0; i<len; i++)
        {
        //idx 表示当前线程需要去取的候选值、其指针偏移的大小。候选值将与其他线程候选值共同筛选出最小值。
        //每个线程负责的数据之间的间隔是 threadnum，而不同线程所取的数应该是相邻的。
        //如果当前线程已经上报了 tidHeader[tid]-1 个数,那么就应该去取该线程的第 tidHeader[tid]个数，
        //加之线程负责的两数之间相距 threadnum 个空间，则第 tid 个线程应该取第 tidHeader[tid] *
threadnum + tid 个数。
        unsigned int idx = tidHeader[tid] * threadnum + tid;
        if(idx < len)
        {
            value = tmp[idx]; //如果指针偏移量小于数据大小，表示没有下标越界，则候选值 value 从相应
空间中取得。
        }                   //否则在 else 分支中被设置为无穷大。
        else
        {
            value = inf;
        }
           //最小值存放空间初始化。这里只有 tid 为 1 的线程初始化，其它线程阻塞至__syncthreads();
        if(tid == 0)
        {
            minValue = inf;
            mintid = inf;
        }
        __syncthreads();
        //将所有线程的候选值进行比较，选出候选值。
        //这里 atomicMin 为原子操作，会将所有线程的值一一串行的与 minValue 进行比较，留下最小值。
        //由于 minValue 为共享内存，原子操作速度会非常快，反之如果是对全局内存利用原子操作，将会导致速
度性能下降。
        atomicMin(&minValue,  value);
        __syncthreads();
        //判断最终的最小值是否为本线程提供的候选值。
        if(minValue == value)
        {
            //如果是，且有多个线程共同提供了最小值，那么选择线程编号最小的那个线程编号作为结果，并上报
存储。
            atomicMin(&mintid, tid);
        }
        __syncthreads();
        //判断是否当前线程就是那个被上报的线程，
        if(min_tid == tid)
        {
            //如果是，代表该线程负责的数据部分有一个值成功归并，那么将指针偏移后移一个。
            tidHeader[tid]++;
```

```
            //同时把候选值插入输出的第 i 个偏移位置。
            data[i] = value;
        }
        }

    }

    __global__ void sortkernel(ubit32* dp_num, ubit32* dp_tmp, unsigned int len)
    {
        unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int sizeperthrd = (len + blockDim.x - 1) / blockDim;
        radix_sort(dp_data, dp_tmp, sizeperthrd, blockDim.x, len, tid);
        mergeparallel(dp_data, dp_tmp, len, tid, blockDim.x);
    }


    // 主入口函数。
    // hp_nums 为输入数组的头指针
    // len 为数组的长度
    void GPUSort(
                  ubit32* hp_nums,
                  unsigned int len
              )
    {

        ubit32* dp_nums = NULL;
        ubit32* dp_tmp = NULL;
        unsigned int gridsize = (size+block.x-1) / block.x;
        dim3 block(BLOCKSIZE,1);
        dim2 grid(gridsize,1);
        cudaMalloc(dp_nums, len*sizeof(ubit32));
        cudaMalloc(dp_tmp, len*sizeof(ubit32));
        //采用 stream 的方式，实现多流并发，重叠不同进程的显存拷贝与核函数运行的时间
        cudaMemcpyAsync(dp_nums, hp_nums, len*sizeof(ubit32), cudaMemcpyHostToDevice,
stream);
        //基数排序
        sortkernel<<<grid,block,0,stream>>>(dp_nums,dp_tmp,add);
        cudaMemcpyAsync(hp_nums, dp_nums, len*sizeof(ubit32), cudaMemcpyDeviceToHost,
stream);
        cudaDeviceSychronize();
        cudaFree(dp_nums);
        cudaFree(dp_tmp);
        //以下步骤进行如下操作：
        //1、基数排序的结果只是将每个 block 排序为了有序序列，但是 block 之间仍需要进行有序序列归并。
        //2、为了进行归并，采用一个偏移计数数组，分别记录每个 block 数据已经完成了归并的偏移量。
        //3、归并需要 O(N)时间复杂度和 O(N)的空间复杂度。
        vector<unsigned int> blkidHeader(gridsize,0);
        vector<unsigned int> temp;
        for(int i=0; i < len; i++)
        {
            ubit32 minval = MAX_UINT;
            for(int j = 0; j < gridsize; j++)
            {
                if(minval > hp_nums[j * gridsize + blkidHeader[j]])
                {
                    minval = hp_nums[j * gridsize + blkidHeader[j]];
                    minid  = j;
                }
            }
            blkidHeader[j]++;
            temp[i] = minval;
        }
        for(int i=0; i < len; i++)
        {
            hp_nums[i] = temp[i];
        }
```

```
    }
```

4. 附录：

4.1.算法原理：

4.1.1. 串行基数排序步骤如下：

1、取所有数字的二进制形式。

2、初始为：最先比较所有值的最低位。

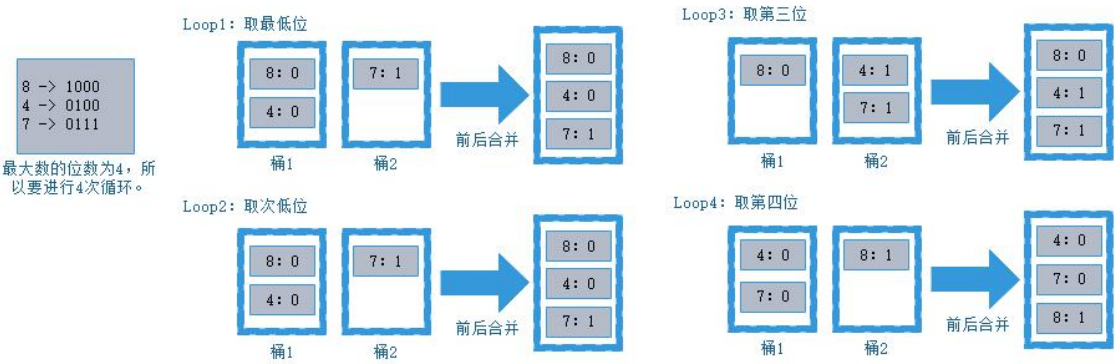
3、开始循环：

(1) 把该位为 0 的数字放入表示序列前部分的数组中。

(2) 把该位为 1 的数字放入表示序列后部分的数组中。

(3) 考察更高一位，重复(1)。

以一个包含[8 4 7]三个数的数组为例，基数排序步骤如下：



4.1.2. GPU 基数排序原理：

GPU 的基数排序原理与 CPU 版本基本类似，但其中的区别在于：

1、每个线程每次循环不再取相邻的数，而是取一定跨度的位置，为的是保证合并访问。

2、桶的储存方式也不再是一个线程将该线程负责的下一个数放入桶的位置。而是同样一定跨度的位置。

3、为了保证不会频繁的访问全局内存，需要将桶定义为共享内存，供 Block 中所有线程访问。

4.1.3. Block 内有序序列归并：

这一步操作的目的在于将各线程分别有序的序列，最终合并成在一个 Block 之内全局有序的序列。因为如上的 GPU 基数排序仅仅保证了每个线程所管理的区域中的数据彼此有序（每个线程管理的区域相邻有一定的跨度），而数组内相邻的数据是由不同的线程管理的。所以现在相邻的数据之间是无序的，且有一定跨度的。而被同一个线程管理的一定跨度空间之间的数据是有序的，所以要做归并操作。

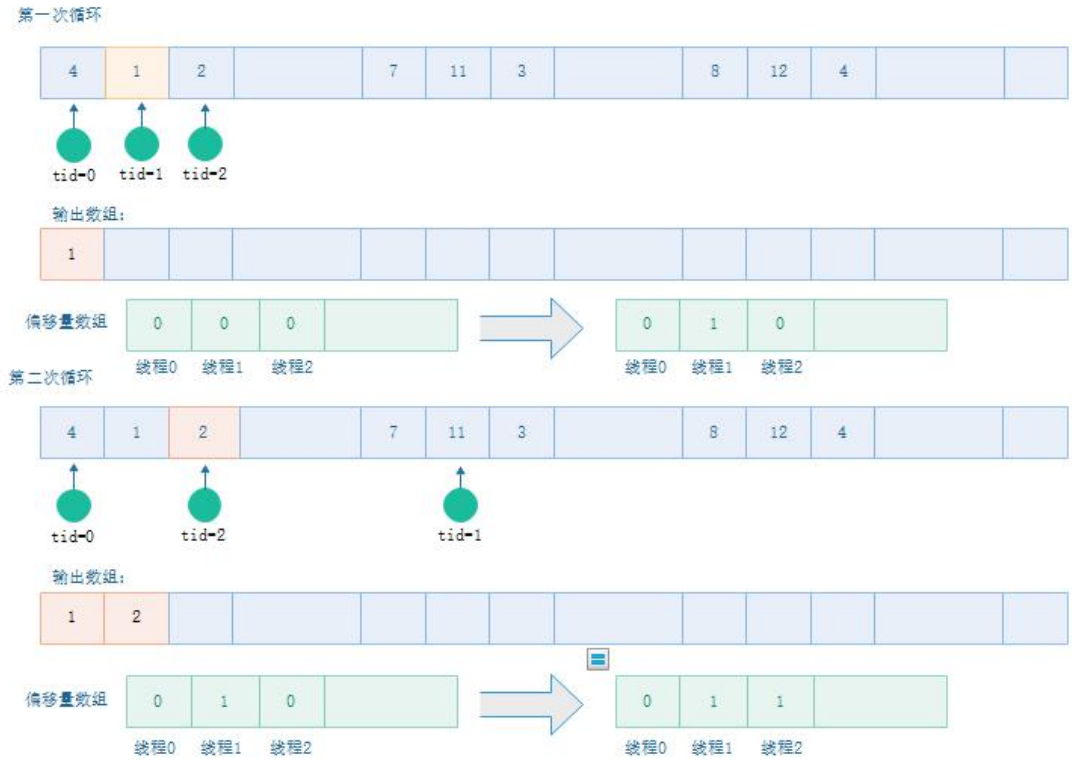
归并操作的方式是：

1、建立一个与**线程**数量一样大小的数组，称为**偏移量数组**。该数组管理了每个线程需要通过多少偏移，找到自己所管理数据中的当前候选数字。除此之外，**偏移量数组**的另一个含义是：目前每个线程已经上报了多少自己管理的数组了。

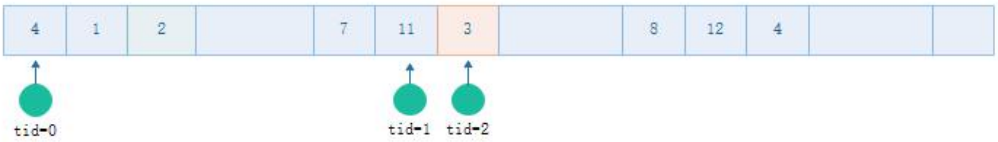
2、循环与整体数组大小相同的次数，每次循环都要找到每个线程管理的位置中最小的那个数字。

3、确定了那个最小的数字后，将它拷贝入结果显存空间，并记录此数字属于哪一个线程。然后到**偏移量数组**内，将对应线程编号的下标空间内，将偏移量自加 1。

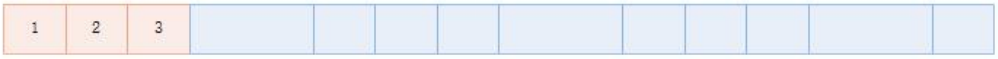
4、进入下一个循环，回到步骤 2。



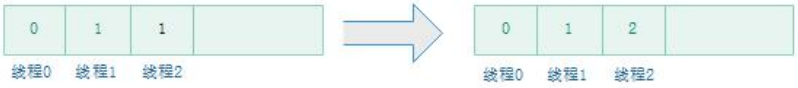
第三次循环



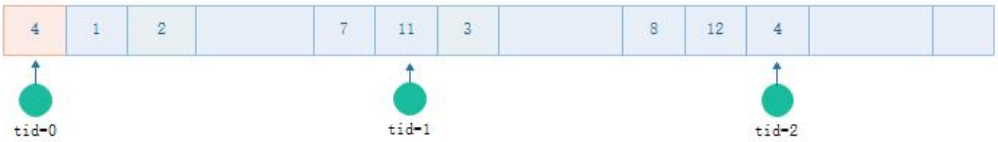
输出数组：



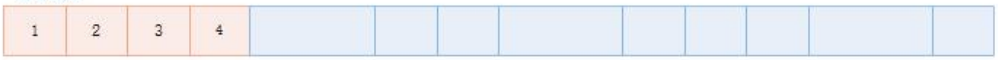
偏移量数组



第四次循环



输出数组：



偏移量数组